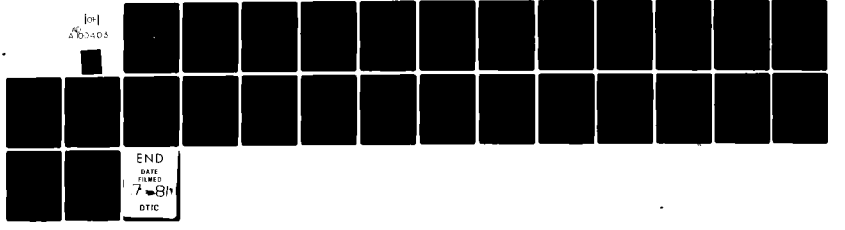
AD-A100 403 OFFICE OF THE UNDER SECRETARY OF DEFENSE FOR RESEARCH--ETC F/G 9/2
DEPARTMENT OF DEFENSE REQUIREMENTS FOR HIGH ORDER COMPUTER PROG--ETC(U)
JAN 77

UNCLASSIFIED NL



END
DATE FILMED
7-81
DTIC

AD A100403

DEPARTMENT OF DEFENSE

REQUIREMENTS FOR HIGH ORDER

COMPUTER PROGRAMMING LANGUAGES

DTIC
ELECTE
JUN 19 1981
S
A

"IRONMAN"

14 January 1977

This document has been approved for public release and sale; its distribution is unlimited.

81 6 19 002

OFFICE OF THE UNDER SECRETARY OF DEFENSE
WASHINGTON, D.C. 20301

RESEARCH AND
ENGINEERING

Dear Friend of Ada:

Thank you for your interest in Ada.

Your name has been added to the Ada mailing list, and occasionally you will receive information from the Ada Joint Program Office concerning the status of the Ada program.

Under the Freedom of Information Act, the Ada Joint Program Office (AJPO) mailing list is being made available on the USC-ECLB computer. If you object to inclusion of your name on this public list, please inform the AJPO in writing. To help keep the list up-to-date, please notify the AJPO of address changes.

Sincerely,

for Larry E. Druffel, Lt. Col., USAF
Director, Ada Joint Program Office

Accession For
NTIS GRA&I
DTIC TAB
Unannounced
Justification

1

THE TECHNICAL REQUIREMENTS

The technical requirements for a common DoD high order programming language given here are a synthesis of the requirements submitted by the Military Departments. They specify a set of language characteristics that are appropriate for embedded computer applications (i.e., command and control, communications, avionics, shipboard, test equipment, software development and maintenance, and support applications).

The changes that produced this revision reflect the many comments on previous versions received from the Services, military contractors, the research community, and other organizations during 1976. This revision does not alter the basic intent or substance of the December 1975 (i.e., "TINMAN") version of the requirements. It does incorporate changes to improve clarity, to correct errors, and to ensure feasibility.

The revised requirements are hierarchically organized with an outline similar to that expected in a language defining document. Section 1 gives the general design criteria. These provide the major goals that influenced the selection of specific requirements and provide a basis for language design decisions that are not otherwise dealt with in the requirements. Section 2 through 12 give more specific technical requirements on the language and its translators. The requirements call for the inclusion of features to satisfy specific needs in the design, implementation, and maintenance of military software, specify many general and specific characteristics desired for the language, and call for the exclusion of certain undesirable characteristics. Section 13 gives some of the intentions and expectations for development, control, and use of the language. The intended use and environment for the language has strongly influenced the requirements; understanding those intentions should aid in achieving the requirements.

A precise and consistent use of terms has been attempted throughout the requirements. Potentially ambiguous terms have been defined in the text. Care has been taken to distinguish between requirements, given as text, and comments about the requirements, given as bracketed notes. Previously duplicative requirements have been given just once. Potentially conflicting implications of requirements have been clarified.

The following terms have been used throughout the text to indicate where and to what degree individual requirements apply:

shall	indicates a requirement placed on the language or translator
should	indicates a desired goal but one for which there is no objective test
shall attempt	indicates a desired goal but one that may not be achievable given the current state of the art, or may be in conflict with other more important requirements
shall require	indicates a requirement that is to be placed on the user by the language and its translators
shall permit	indicates a requirement placed on the language to provide an option to the user
must	same meaning as shall require but takes user as subject
may	same meaning as shall permit but takes user as subject
will	indicates a consequence that is expected to follow or indicates an intention of the DoD; it does not in any case by itself constrain the design of the language.

Some of the above terms are also used informally in the bracketed notes. Language is used in the singular to refer to the minimum number of languages necessary to satisfy the needs of DoD applications.

1. General Design Criteria

1A. Generality. The language shall provide generality only to the extent necessary to satisfy the needs of embedded computer applications. Such applications require real time control, self diagnostics, input-output to nonstandard peripheral devices, parallel processing, numeric computation, and file processing. The language shall not contain features that are unnecessary to satisfy the requirements.

1B. Reliability. The language should aid the design and development of reliable programs. The language shall be designed to avoid error prone features and to maximize automatic detection of programming errors. The language shall require some redundant, but not duplicative, specifications in programs. Translators shall produce explanatory diagnostic and warning messages, but shall not attempt to correct programming errors.

1C. Maintainability. The language should promote ease of program maintenance. It should emphasize program readability over writability. That is, it should emphasize the clarity, understandability, and modifiability of programs over programming ease. The language should encourage user documentation of programs. It shall require explicit specification of programmer decisions and shall provide defaults only for instances where the default is stated in the language definition, is always meaningful, reflects common usage, and can be explicitly overridden.

1D. Efficiency. The language design should aid the production of efficient object programs. Constructs that have exceptionally expensive or exceptionally inexpensive implementations should be easily recognizable by translators and by users. Users shall be able to specify the time space trade offs in a program. Where possible, features should be chosen to have a simple and efficient implementation in many object machines, to avoid execution costs for available generality where it is not needed, to maximize the number of safe optimizations available to translators, and to ensure that unused and constant portions of programs will not add to execution costs. Execution time support packages of the language shall not be included in object code unless they are called.

1E. Simplicity. The language should not contain unnecessary complexity. It should have a consistent semantic structure that minimizes the number of underlying concepts. It should be as small as possible consistent with the needs of the intended applications. It should have few special cases and should be composed from features that are individually simple in their semantics. The language should have uniform syntactic conventions and should not provide several notations for the same concept.

1F. Implementability. The language shall be composed from features that are understood and can be implemented. The semantics of each feature should be sufficiently well specified and understandable that it will be possible to predict its interaction with other features. To the extent that it does not interfere with other requirements, the language shall facilitate the production of translators that are easy to implement and are efficient during translation. There shall be no language restrictions that are not enforceable by translators.

1G. Machine Independence. The language shall strive for machine independence. It shall not dictate the characteristics of object machines or operating systems. The design of the language shall attempt to avoid features whose semantics depend on characteristics of the object machine or of the object machine operating system. There shall be a facility for specifying those portions of programs that are dependent on the object machine configuration and for conditionally compiling programs depending on the actual configuration.

1H. Formal Definition. To the extent that a formal definition assists in achieving the above goals, the language shall be formally defined. [Note that formal definitions are of most value during language design; and that the same method may not be appropriate for defining all aspects of a language.]

2. General Syntax

2A. Character Set. Every construct of the language shall have a representation that uses only the 64 character subset of ASCII:

```
 !"#$%&'()*+,-./
0123456789:;<=>?
@ABCDEFGHIJKLMNO
PQRSTUVWXYZ[\]^_
```

2B. Grammar. The language should have a simple, uniform, and easily parsed grammar and lexical structure. The language shall have free form syntax and should use familiar notations where such use does not conflict with other goals.

2C. Syntactic Extensions. The user shall not be able to modify the source language syntax. In particular the user shall not be able to modify or introduce new precedence rules or to define new syntactic forms.

2D. Other Syntactic Issues. Multiple occurrences of a language defined symbol appearing in the same context shall not have essentially different meanings. The language shall not permit unmatched parentheses of any kind (e.g., begin and end must be paired one for one). Source program line boundaries shall be treated like spaces. All key word forms that contain declarations or statements shall be bracketed (i.e., shall have a closing as well as an opening key word).

2.1. Identifiers

2E. Mnemonic Identifiers. Mnemonically significant identifiers shall be allowed. There shall be a break character for use within identifiers. The language and its translators shall not permit identifiers or reserved words to be abbreviated.

2F. Reserved Words. The only reserved words shall be those that introduce special syntactic forms or that are otherwise used as delimiters. Words that can be used in place of identifiers shall not be reserved (e.g., names of built-in or predefined functions, types, constants, and the like shall not be reserved). All reserved words shall be listed in the language definition.

2.2. Literals

2G. Numeric Literals. There shall be built-in numeric literals. Numeric literals shall have the same values in programs as in data.

2H. String Literals. There shall be built-in string literals. String literals shall be interpreted as fixed length one-dimensional character arrays. Literal strings shall not be allowed to cross line boundaries of the source program.

2.3. Comments

2I. Comments. The language shall allow comments to be embedded within program text (e.g., a comment bracketed by special left and right bracket symbols) and shall allow stand alone comments (e.g., a comment introduced by a special symbol at the beginning of each line). Bracket symbols shall consist of no more than two characters each. The language shall not permit comments to automatically cross line boundaries.

3. Types

3A. Strong Typing. The language shall be strongly typed. That is, the type or mode of each variable, array and record component, expression, function, and parameter shall be determinable at translation time.

3B. Implicit Type Conversions. There shall be no implicit conversions between types.

3C. Type Definitions. It shall be possible to define new data types in programs. Type definitions shall be processed entirely at translation time. The scope of a type definition shall be determinable at translation time. No restriction shall be imposed on defined types unless it is imposed on all types.

3.1. Numeric Types

3-1A. Numeric Values. *The language shall provide types for integer, fixed point,* and floating point numbers. Numeric operations and assignment that would cause the most significant digits of numeric values to be truncated (e.g., when overflow occurs) shall constitute an exception situation.

3-1B. Numeric Operations. There shall be built-in operations (i.e., functions) for conversion between numeric types. There shall be built-in operations for addition, subtraction, multiplication, division with floating point result, and negation for all numeric types. There shall be built-in equality (i.e., equal and unequal) and ordering operations (i.e., less than, greater than, less or equal, and greater or equal) between elements of each numeric type. Numeric values shall be equal if and only if they represent exactly the same abstract value. The semantics of all built-in numeric operations shall be included in the language definition. [Note that there might also be standard library definitions for numeric functions such as exponentiation.]

3.1.1. Floating Point Type

3-1C. Floating Point Precision. The precision of each floating point variable and expression shall be specifiable in programs and shall be determinable at translation time. Precision specifications shall be required for each floating point variable. Precision shall be interpreted as the minimum precision to be implemented in the object machine. Floating point results shall be implicitly rounded (or on some machines truncated) to the implemented precision. Explicit conversion operations shall not be required between floating point precisions.

3-1D. Floating Point Implementation. A floating point computation may be implemented using the actual precision, radix, and exponent range available in the object machine hardware. There shall be built-in operations to access the actual precision, radix, and exponent range with which floating point variables and expressions are implemented.

3.1.2. Integer and Fixed Point Types

3-1E. Integer and Fixed Point Numbers. Integer and fixed point numbers shall be treated as exact numeric values. There shall be no implicit truncation or rounding in integer and fixed point computations.

3-1F. Integer and Fixed Point Variables. The range of each integer and fixed point variable must be specified in programs and determinable at translation time. Such specifications shall be interpreted as the minimum range to be implemented. Explicit conversion operations shall not be required between numeric ranges.

3-1G. Fixed Point Scale. The scale or step size (i.e., the minimal representable difference between values) of each fixed point variable must be specified in programs and be determinable at translation time.

3-1H. Integer and Fixed Point Operations. There shall be built-in operations for integer and fixed point division with remainder and for conversion between fixed point scale factors. The language shall require explicit scale conversion operations whenever the scale of a value must be changed to properly perform some operation (e.g., assignment, comparison, or parameter passing).

3.2. Enumeration Types

3-2A. Enumeration Type Definitions. There shall be types that are definable in programs by enumeration of their elements. The elements of an enumeration type may be identifiers or character literals. Literal identifiers shall be syntactically distinguishable from other identifiers. Equality and inequality shall be automatically defined between elements of each enumeration type.

3-2B. Ordered Enumeration Types. Ordered enumeration types must be so marked in their definitions. The four ordering operations shall be automatically defined between elements of each ordered type defined by enumeration. A variable of an ordered enumeration type may be restricted to a contiguous subsequence of the enumeration.

3.2.1. Boolean Type

3-2C. Boolean Type. There shall be a predefined unordered enumeration type for Boolean values. The Boolean type shall have operations for conjunction, inclusive disjunction, and negation.

3.2.2. Character Types

3-2D. Character Types. Character sets shall be definable as enumeration types. Character types may contain both printable and control characters. Definitions for ASCII and other widely used character sets shall be available in a standard library.

3.3. Composite Types

3-3A. Composite Type Definitions. It shall be possible to define types that are Cartesian products of other types. Composite types shall include arrays (i.e., composite data with indexible components of homogeneous types) and records (i.e., composite data with labeled components of heterogeneous type).

3-3B. Component Specifications. For elements of composite types, the type of each component (i.e., field) must be explicitly specified in programs and determinable at translation time. Components may be of any type (including array and record types). Range, precision and scale specifications shall be required for each component of appropriate numeric types.

3-3C. Operations on Composite Types. A value accessing operation shall be automatically defined for each component of composite data elements. Assignment shall be automatically defined for components that have alterable values. A constructor operation (i.e., an operation that constructs an element of a type from its constituent parts) shall be automatically defined for each composite type. An assignable component may be used anywhere in a program that a variable of the component's type is permitted.

3.3.1. Arrays

3-3D. Array Specifications. The number of dimensions for each array must be specified in programs and shall be determinable at translation time. The range of subscript values for each dimension must be specified in programs and shall be determinable by the time of array allocation. The range of subscript values shall be restricted to a contiguous sequence of integers or to a contiguous sequence from an enumeration type. [Note that translators may be able to produce more efficient object programs where subscript ranges are determinable at translation time.]

3-3E. Operations on Subarrays. There shall be built-in operations for value access, assignment, and catenation of contiguous sections of one-dimensional arrays of the same component type.

3.3.2. Records

3-3F. Operations on Records. Assignment shall be permitted between records with corresponding components of identical name and type.

3-3G. Nonassignable Record Components. It shall be possible to specify record components (including tag fields) for which assignment shall not be permitted. These components shall include those defined as constants and those defined as expressions. [Note that such components need not take data storage space.]

3-3H. Variant Types. It shall be possible to define types with alternative record structures (i.e., variants). The structure of each variant shall be determinable at translation time. Each variant must have a tag field (i.e., component that can be used to discriminate among the variants during execution). The value of a variant may be used anywhere a value of the variant type is permitted.

3.3.3. Types Requiring Dynamic Allocation

3-3I. Definitions of Dynamic Types. It shall be possible to define types whose elements are dynamically allocated. Elements of such types may have components of their own type and may have substructure that can be altered during execution. Such types shall be distinguishable from other composite types in their definitions. [Note that such types require pointers and heap storage in their implementations. They are intended primarily for the support portions of embedded computer software.]

3-3J. Constructor Operations. Each execution of the constructor operation for a dynamically allocated type shall create a distinct element of the type. Such elements shall remain allocated as long as there is an access path to them.

3.4. Set Types

3-4A. Set Type Definitions. It shall be possible to define types as power sets of enumeration types. [Note that the elements of such types are sets and can be implemented as bit strings.]

3-4B. Operations on Sets. Membership and constructor operations shall be defined automatically for each type defined as a power set. Intersection, union, symmetric difference, equality, and inequality shall be automatically defined between elements of each set type. [Note that intersection, union, and symmetric difference can be implemented as bit by bit operations for conjunction, inclusive disjunction, and exclusive disjunction, respectively.]

3.5. Encapsulated Types

3-5A. Encapsulated Definitions. It shall be possible to encapsulate definitions. Encapsulations may contain definitions of the data elements comprising a type and of operations.

3-5B. Effect of Encapsulation. The effect of encapsulation shall be to inhibit external access to implementation properties of the definition. In particular declarations made within an encapsulation shall not automatically be accessible outside the encapsulation. Data elements defined in an encapsulation shall not automatically inherit the operations of the types with which they are represented.

3-5C. Own Variables. It shall be possible within encapsulations to declare variables that are accessible only within the encapsulation but remain allocated throughout the scope in which the encapsulation is declared. Such variables shall retain their values between entries to the encapsulation. It shall be possible to initialize such variables at the time of their apparent allocation.

3-5D. Operations Between Types. It shall be possible to define operations, like type conversion, that require access to local properties of more than one encapsulated definition. [Note that this capability violates the purpose of encapsulation and thus its use should be avoided wherever possible.]

4. Expressions

4A. Form of Expressions. The form (i.e., context free syntax) of expressions shall not depend on the types of their operands or on whether the types of the operands are built into the language.

4B. Type of Expressions. The language shall require that the type of each expression be determinable at translation time. It shall be possible to specify the type of an expression explicitly. [Note that the latter requirement provides a way to resolve ambiguities in the types of literals and to assert the type of results; it does not provide a mechanism for type conversion.]

4C. Side Effects. The language should permit few side effects in expressions. In particular, during expression evaluation assignment shall not be allowed to any variable that is accessible in the scope of the expression.

4D. Allowed Usage. Expressions of a given type shall be allowed wherever both constants and variables of the type are allowed.

4E. Constant Valued Expressions. Constant valued expressions (i.e., expressions whose values are determinable at translation time) shall be allowed wherever constants of the type are allowed. Such expressions shall be evaluated before execution time.

4F. Operator Precedence Levels. The precedence levels (i.e., binding strengths) of all infix operators shall be specified in the language definition, shall not be alterable by the user, shall be few in number, (e.g., three or four), and shall not depend on the types of the operands. [Note that there might be built-in operator symbols whose meaning is entirely specified by the user.]

4G. Effect of Parentheses. Explicit parentheses shall dictate the association of operands with operators. Explicit parentheses shall be required to resolve the operator-operand associations wherever an expression has a nonassociative operator to the left of an operator of the same precedence.

5. Constant, Variables, and Declarations

5A. Declarations of Constants. It shall be possible to associate identifiers with constant values of any type that is not dynamically allocated. Constants shall include both those whose values are determinable at translation time and those whose value cannot be determined until scope entry time. A translation time error shall be reported whenever a program attempts to assign to a constant valued identifier.

5B. Declarations of Variables. There shall be no default declarations for variables. The type of each variable must be explicitly specified in programs and shall be determinable at translation time. Variables may be of any type.

5C. Scope of Declarations. The intended scope of a declaration shall be determinable from the program at translation time. Scopes may be lexically embedded. Translators shall provide a warning wherever a local definition masks a more global definition. [Note that a function need not mask a more global function if they differ in name, number of parameters, or formal parameter types.]

5D. Restrictions on Values. Procedures, functions, types, labels, exception situations, and statements shall not be assignable to variables, computable as values of expressions, or usable as parameters to procedures or functions.

5E. Initial Values. There shall be no default initial values for variables. The same syntactic form shall not be used both to declare constants and to initialize variables. [Note that initialization of variables must (except for some global variables) be accomplished during execution, not translation.]

5F. Operations on Variables. Assignment and an implicit value access operation shall be automatically defined for each variable.

5G. Other Declarations. It shall be possible to associate identifiers with specifications of type and representation (including range, scale, and precision). Such identifiers may be used in declarations of variables, to specify components of elements of composite types, and in formal parameter specifications.

6. Control Structures

6A. Basic Control Facility. The built-in control mechanisms should be of minimal number and complexity and where possible shall be structured (i.e., shall have one point of entry and shall exit to a single point). Each shall provide a single capability and shall have a distinguishing syntax. Nesting of control structures shall be allowed. There shall be no control definition facility. Local scopes shall be allowed within the bodies of control statements.

6B. Sequential Control. There shall be a sequential control mechanism (i.e., a mechanism for sequencing statements). Explicit statement delimiters shall be required. [Note the choice between terminators and separators can be left to the user.]

6C. Conditional Control. There shall be conditional control structures that permit selection among alternative control paths. The selected path may depend on the value of a conditional expression, on a computed choice among labeled alternatives, or on the true condition in a set of mutually exclusive conditions. The control action must be specified for all values of the discriminating condition. [Note that only one branch will be compiled when the selected case for a conditional statement is determinable at translation time.]

6D. Short Circuit Evaluation. There shall be forms for short circuit conjunction and disjunction of Boolean expressions in conditional and iterative control structures.

6E. Iterative Control. There shall be an iterative control structure that permits a loop to have several explicit termination conditions and permits termination anywhere in the loop. Iterative control structures may be entered only at the head of the loop. [Note that when the number of iterations is zero or one and is determinable at translation time, the translator can omit any unnecessary object code.]

6F. Loop Control Variables. Loop control variables, if any, shall be local to the iterative control statement. Assignment shall not be allowed to control variables from the loop body. It shall be possible to iterate over sequences of integers and over elements of an enumeration type.

6G. Explicit Control Transfer. There shall be an explicit mechanism for control transfer (i.e., the go to). The go to shall not permit transfer of control out of declarations (including functions, procedures, and encapsulated definitions) or out of parallel control structures. It shall not permit transfer into narrower access scopes or into control structures (e.g., conditional, iterative, and parallel control structures). There shall be no control transfer mechanisms in the form of switches, designational expressions, label variables, label parameters, or alter statements.

7. Functions and Procedures

7A. Function and Procedure Definitions. Functions (which return values to expressions) and procedures (which can be called as statements) shall be definable in programs. Existing functions (including those called using infix forms) and procedures shall be extendible to new data types (i.e., overloading shall be permitted).

7B. Recursive Definitions. It shall be possible to define functions and procedures recursively as well as nonrecursively. If necessary the language shall restrict recursive definitions to insure that they do not add to the execution cost of other program constructs.

7C. Scope Rules. A reference to an identifier (other than an identifier for an exception situation) that is not declared in the most local scope shall refer to a program element that is lexically global, rather than to one that is global through the dynamic calling structure.

7.1. Functions

7D. Function Declarations. The result type for each function must be explicitly specified in the function declaration and shall be determinable at translation time. A function of two arguments may be specified as associative in its declaration. [Note that the latter requirement reduces the need for explicit parentheses.]

7E. Restrictions on Functions. A function may only have input parameters and may not be called in a scope that contains variables that are referenced or assigned directly or indirectly within the body of the function. [Note that this requirement guarantees that parameters to functions can be implemented safely with either value or reference passing.]

7.2. Parameters

7F. Formal Parameter Classes. There shall be three classes of formal parameters: 1) input parameters, which act as constants that are initialized to the value of corresponding actual parameters at the time of call, 2) input-output parameters, which enable access and assignment to the corresponding actual parameters, and 3) output parameters, which act as local variables whose values are transferred to the corresponding actual parameter only at the time of normal exit. In the latter two cases the corresponding actual parameter must be a variable or an assignable component of a composite type.

7G. Parameter Specifications. The type of each formal parameter must be explicitly specified in programs and shall be determinable at translation time. Parameters may be of any type. Range, precision, and scale specifications shall be required for each formal parameter of appropriate numeric types. A translation time error shall be reported wherever corresponding formal and actual parameters are of different types and wherever a program attempts to use a constant or an expression where a variable is required.

7H. Formal Array Parameters. The number of dimensions for formal array parameters must be specified in programs and shall be determinable at translation time. Determination of the subscript range for formal array parameters may be delayed until execution and may vary from call to call. Subscript ranges shall be accessible within function and procedure bodies without being passed as an explicit argument.

7I. Restrictions to Prevent Aliasing. Aliasing (i.e., multiple access paths to the same variable from a given scope) shall not be permitted. In particular, a variable may not be used as two output arguments in the same call to a procedure, and a nonlocal variable that is accessed or assigned within a procedure body may not be used as an output argument to that procedure.

8. Input-Output Facilities

8A. Low Level Input-Output Operations. There shall be a set of built-in low level input-output operations that act on physical files (e.g., input-output channels and peripheral devices). The low level operations shall be chosen to insure that all application level input-output operations can be defined within the language. They shall include operations to send control information, to receive control information, to begin transfer of data in either direction, and to wait for completion of a data transfer.

8B. Application Level Input-Output Operations. There shall be standard library definitions for application level input-output to logical files. These shall include operations for creating, deleting, opening, closing, reading, writing, and positioning logical files. The meaning of such operations shall depend on the general characteristics of the files or devices (e.g., on whether they are sequentially or randomly accessed), but shall not be dependent on a specific device.

8C. Input Restrictions. Input shall be restricted to files whose data representation is known to the translator (i.e., to files that are created and written entirely within the program or to files whose data representation is explicitly specified in the program).

8D. Operating System Independence. The language shall not require the presence of an operating system. The form and meaning of built-in and library definitions shall not be dependent on the operating system, if present. [Note that functions and operators of the language can be implemented as operating system calls where the operating system is compatible with the function or operator definition.]

8E. Configuration Control. There shall be a few low level facilities that permit programs (usually library routines) to interrogate and control the status of physical resources (e.g., memory or processors) that are managed (e.g., allocated or scheduled) by built-in features of the language. In particular it shall be possible to dynamically reassign the association between physical and logical devices, to control program overlays, and to prevent allocation and scheduling of faulty resources.

9. Parallel Processing

9A. Parallel Control Structures. There shall be a control structure for parallel processing. It shall permit a fixed number (i.e., determinable at translation time) of control paths to operate in parallel and to rejoin at a single point. There shall be an operation that is executable on any path of a parallel control structure and that causes immediate termination of the other paths (i.e., causes the other paths to move to the rejoin point).

9B. Parallel Path Implementation. The parallel processing facility shall be designed to minimize execution cost. In particular, parallel control paths shall be implementable with multiprocessors or with interleaved execution on a single processor.

9C. Mutual Exclusion. There shall be a mechanism for mutual exclusion among parallel processes. During specified portions of its execution, a parallel path shall be able to wait for and gain exclusive use of certain program declared objects and to release those objects. [Note that special asynchronous hardware and software interrupt facilities are not necessary in the language; interrupts can be treated as objects that are released upon occurrence of the interrupt.]

9D. Real Time Constraints and Scheduling. Constraints on the real (i.e., elapsed) time for execution of portions of control paths shall be specifiable in programs. Translators shall give warning if there is risk that time constraint will not be met. It shall also be possible to specify which paths are to be given preference in execution (in case the number of actual paths exceeds the number of available processors or the processors execute at different speeds). [Note: Such specifications provide a means to document the real time constraints of the applications, but do not specify a specific execution order among parallel paths and do not provide a safe means for mutual exclusion.]

9E. Real Time Clock. There shall be an accessible real time clock. It shall be possible to specify a delay on any control path for specified real time intervals. Such specifications shall be interpreted as the minimum time before continuing execution on that control path.

9F. Simulated Time Clock. There shall be an accessible simulated time clock. It shall be possible to delay any control path for a specified simulated time interval.

10. Exception Handling

10A. Exception Handling Facility. There shall be an exception handling mechanism for responding to unplanned error situations detected during program execution. The exception situations shall include errors detected by hardware, software errors detected during execution, error situations in built-in operations, and attempts to execute portions of programs that are not present in main memory. Exceptions should add to the execution time of programs only if they are invoked.

10B. Software Error Situations. The software errors detectable during execution shall include exceeding the specified range of an array subscript, exceeding the specified range of a variable, exceeding the implemented range of a variable, attempting to access an uninitialized variable, and failing to satisfy a program specified assertion. [Note that many range checks can be done during translation thereby reducing execution costs.]

10C. Invoking Exceptions. During any function or procedure execution it shall be possible to invoke an exception situation in the calling statement. This exception shall cause termination of the routine and an immediate transfer of control in the caller. Such exceptions must be specified in the definition of the function or procedure. Exceptions that can be invoked by built-in operations shall be given in the language definition.

10D. Processing Exceptions. There shall be a control structure for discriminating among the exceptions that can occur in a specified portion of a program. Exceptions that are not processed at a given function or procedure level shall cause termination of the function or procedure and shall invoke an exception in its caller. Exceptions that cause exit from parallel control structures shall terminate all paths of the parallel control structure.

10E. Order of Exceptions. The order in which exceptions in different parts of an expression are detected shall not be guaranteed by the language or by the translator.

10F. Assertions. It shall be possible to include assertions in programs. If an assertion is false when encountered during execution, it shall invoke an exception. [Note that assertions can also be used to aid optimization and maintenance.]

10G. Suppressing Exceptions. It shall be possible to suppress individually the detection of exceptions for software error situations. Should such a situation occur when its detection is suppressed, the consequences will be unpredictable.

11. Specifications of Object Representation

11A. Data Representation. The language shall permit but not require programs to specify the physical representation of data. These specifications shall be distinct from the logical descriptions. Specifications for the order of fields, the width of fields, the presence of "don't care" fields, the positions of word boundaries, and the object representation of atomic data shall be allowed. If object representations are not specified, they shall be determined by the translator.

11B. Multiple Representations. It shall be possible in programs to define more than one physical representation (e.g., packed and unpacked) for elements of a given type, and to associate a specific representation with each variable of that type. [Note that changes of representation can be accomplished through assignment.]

11C. Machine Configuration Constants. The language shall require the declaration of certain global constants of the object machine configuration. These shall include constants that specify the machine model, the memory size, special hardware options, the operating system if present, and peripheral equipment. Such constants shall be used to determine the object code to be generated by the translator and may also be used by the program like other constants. [Note that the user can define constants and use them as switches to control user defined compilation options.]

11D. Configuration Dependent Specifications. It shall be possible to use machine dependent facilities in programs. Portions of programs that depend on the characteristics of the object machine (e.g., on the machine model, special hardware options, device configuration, or operating system) shall be permitted only within branches of conditional control structures that discriminate on the object machine configuration.

11E. Code Insertions. For some object machines it shall be possible to write programs that include encapsulated code written in machine language or in other established programming languages. Such facilities shall be modest and shall attempt to maximize safety. The language should be designed to minimize the need for code insertions.

11F. Optimization Specifications. It shall be possible in programs to specify the optimization criteria to be used. It shall be possible to specify whether minimum translation costs or minimum execution costs are more important. In the latter case the user may also specify whether execution time or memory space is to be given preference. The meanings of program constructs (other than execution time and space) shall not depend on the optimizations that are applied.

12. Library, Separate Compilation, and Generic Definitions

12A. Library Entries. The language shall support the use of an external library of definitions and separately compiled segments. Library entries shall include type definitions, input-output packages, common pools of shared declarations, and application oriented software packages. The library shall be structured to allow entries to be associated with a particular application, project or user.

12B. Separately Compiled Segments. The language shall support the assembly of separately compiled program segments into an operational program. It shall allow definitions made in one separately compiled segment to be used in another, and shall require that such definitions and declarations be explicitly exported from the defining segment and be explicitly, but not necessarily individually, imported to the using segment. Type constraints and other program and language imposed restrictions shall be enforced across such interfaces.

12C. Restrictions on Separate Compilation. Separate compilation shall not change the meaning of a program. Translators shall be responsible for the integrity of object code in affected segments when any segment is modified, and shall insure that shared definitions have compatible representations in all segments. [Note: This suggests that a segment cannot be compiled until all segments from which it imports definitions and declarations, are defined.]

12D. Generic Definitions. It shall be possible to define functions, procedures, and types with parameters that are instantiated during translation at each call. Such parameters may be any defined identifier (including those for variables, functions, or types), an expression, or a statement. These parameters, like all other parameters, shall be evaluated in the context of the call. [Note that generic definitions generally cannot be separately compiled, but where generic definitions are implemented as closed routines, several instantiations can often share the same object code.]

13. Support for the Language

13A. Defining Documents. The language shall have a complete and unambiguous definition. It should be possible to predict the complete action of any syntactically correct program from the language definition. The language documentation shall include the syntax, semantics, and appropriate examples of each feature including those for standard library definitions. The defining documentation might point out the relative efficiency of alternative constructs.

13B. Standards. There will be a standard definition of the language. Procedures will be established for standards control and for certification that implementations meet the standard.

13C. Subset and Superset Implementations. Translators shall implement the standard definition. There shall be no subset or superset implementations. Every feature that is available to the user shall be defined in the standard, in an accessible library, or in the source program.

13D. Translator Diagnostics. Translators shall be responsible for reporting errors that are detectable at translation time and for optimizing object code. Translators shall do full syntax and type checking, shall check that all language imposed restrictions are met, and shall provide warnings of unusually expensive constructs. A representative set of translation time diagnostic and warning messages shall be included in the language definition.

13E. Translator Characteristics. Translators for the language shall be written in the language and shall be able to produce code for a variety of object machines. Where practical, the machine independent parts of translators should be separate from the code generators. Self hosting of translators is desirable, but is not required (i.e., the translator need not be able to run on all the object machines). The internal characteristics of the translator (i.e., the translation method) shall not be dictated by the language definition or standards.

13F. Translation and Execution Restrictions. Translators should fail to compile correct programs only when the program exceeds the resources or capabilities of the intended object machine or when the program requires more resources during the translation than are available on the host machine. Translators shall report an error when a program requires memory, devices, or special hardware that are unavailable in the object machine. Neither the language nor its translators shall impose arbitrary restrictions on language features. That is, they shall not impose restrictions on the number of array dimensions, on the size of data structures, on the size of set types, on the number of identifiers, on the length of identifiers, or on the number of nested parentheses levels unless such restrictions are dictated by the limitations of the host or object machine and are documented in user accessible manuals.

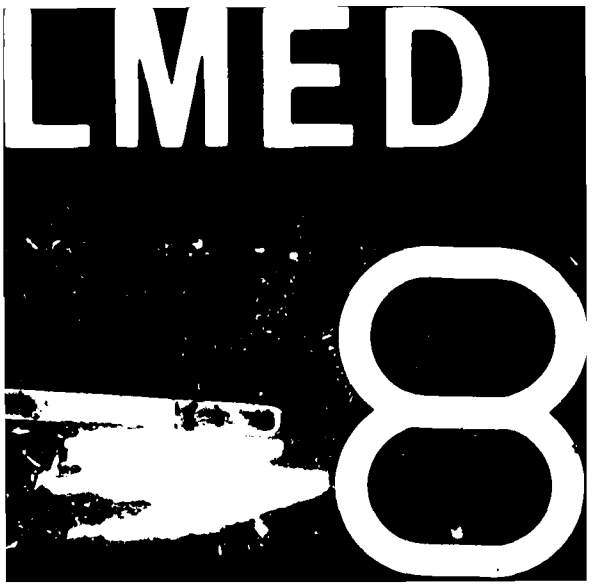
13G. Software Tools and Application Packages. The language shall be designed to work in conjunction with a variety of useful software tools and application support packages. These will be developed as early as possible and will include editors, interpreters, diagnostic aids, program analyzers, documentation aids, testing aids, software maintenance tools, optimizers, and application libraries. There will be a consistent user interface for these tools. Where practical software tools and aids will be written in the language. Support for the design, implementation, distribution, and maintenance of translators, software tools and aids, and application libraries will be provided independently of the individual projects that use them.

For a more detailed discussion of the DoD common language effort, the requirements background, and the relation of programming languages to the DoD software problem see:

1. Department of Defense Requirements for High Order Computer Programming Languages, "TINMAN", June 1976, or

2. IDA Paper P-1191, "A Common Programming Language for the Department of Defense --Background and Technical Requirements", David A Fisher, June 1976.

Both of the these documents have been widely distributed and both contain the December 1975 set of requirements.


LMED
8